A Reliable Message Delivery Protocol for Mobile Agents

M.Ranganathan, Marc Bednarek and Doug Montgomery

Internetworking Technologies Group
National Institute of Standards and Technology
100 Bureau Drive, Gaithersburg, MD 20899.
{mranga, bednarek, dougm}@antd.nist.gov

Abstract. The abstractions and protocol mechanisms that form the basis for inter-agent communications can significantly impact the overall design and effectiveness of Mobile Agent systems. We present the design and performance analysis of a reliable communication mechanism for Mobile Agent systems. Our protocols are presented in the context of a Mobile Agent system called AGNI [1]. We have developed AGNI communication mechanisms that offer reliable peer-to-peer communications, and that are integrated with our agent location tracking infrastructure to enable efficient, failure-resistant networking among highly mobile systems. We have analyzed the design parameters of our protocols using an in-situ simulation approach with validation through measurement of our prototype implementation in real distributed systems. Our system assumptions are simple and general enough to make our results applicable to other Agent systems that may adopt our protocols and/or design principles. [2]

1 Introduction

Mobile Agents are a convenient and powerful paradigm for structuring distributed systems. Using the Agent paradigm, work can be assigned to sequential, event-driven tasks that cooperate with each other to solve a distributed problem. In such systems, Agents roam the network accessing distributed information and resources while solving pieces of the problem. During the course of these computations Mobile Agents need to communicate among themselves to exchange state and status information, control and direct future behavior, and report results.

The abstractions and protocol mechanisms that form the basis for inter-agent communications can significantly impact the overall design and effectiveness of Mobile Agent systems. Numerous approaches to inter-Agent communications are possible including RPC and mailboxes [3]. In this work we present a simple, ordered, reliable, one-way message protocol on top of which other abstractions can easily be built.

Reliable, ordered one-way communication mechanisms greatly simplify the construction of most distributed applications. In traditional distributed applications, TCP [15] provides such services. Through decades of experience and re-engineering, TCP has evolved into a protocol that is highly effective at providing reliable end-to-end data delivery over the conditions found in today's Internet (e.g., link failures, variable latencies, congestion loss).

In this paper, we examine how to build a TCP-like reliable communication mechanism for Mobile Agent systems. The first question to be addressed is "Why we don't use the existing TCP?" We argue that Mobile Agent systems impose new communication requirements and problems that are not adequately addressed by conventional TCP, nor its potential minor variants. In particular, we are concerned with building failure resistant, rapidly re-configurable distributed systems. We view these system properties as a primary motivation for dynamic Agent creation and mobility mechanisms, and as posing significant requirements on inter-Agent communications mechanisms. As such, we require that Agent systems be able to (1) detect and recover from failures in the end-to-end transport mechanism and (2) accommodate efficient communication among mobile end-points. Neither of these capabilities can be provided using standard TCP.

In the remainder of this paper we present the design and performance analysis of a reliable communication mechanism for Mobile Agent systems. Our protocol is presented in the context of a Mobile Agent system called AGNI, whose general design and capabilities have been described earlier [13]. Our communication mechanism

[1] AGNI stands for Agents at NIST and is also Sanskrit for fire.

[2] This work was supported in part by DARPA under the *Autonomous Negotiation Teams* (ANTS) program (AO # 99-H412/00). The work described in this paper is a research project and not an official US. Government endorsement of any product or protocol.

offers reliable peer-to-peer communications that are integrated with our Agent location tracking infrastructure to enable efficient, failure-resistant networking among highly mobile systems. Our system assumptions are simple and general enough to make our results applicable to other Agent systems that may adopt our protocol and design principles. We have analyzed the design parameters of our protocols using an in-situ simulation approach with validation through measurement of our prototype implementation operating in real distributed systems.

The rest of this paper is organized as follows. Section 2 presents an overview of our system design and describes a sliding window protocol for mobile end-points that is the focus of this paper. In Section 3, we present our system simulation design and present evidence that it approximates the real system performance. Section 4 presents results that show the importance of efficient location tracking and message buffering as they relate to the efficiency of the protocol. Section 5 presents related work and finally we conclude with a summary of our findings in Section 6.

2 Mobile Streams and AGNI

We begin by providing an overview of our system model. The specifics of this model are pertinent to our AGNI Mobile Agent system; however, the model is general enough to fit several existing Mobile Agent systems. The basic abstractions, constructs and components of our system are summarized in the following paragraphs.

A Mobile Stream (*MStream*) is a named communication end-point in a distributed system that can be moved from machine to machine while a distributed computation is in progress and while maintaining a sender-defined ordering guarantee of message consumption with respect to the order in which messages are sent to it.

An MStream has a globally unique name. We refer to any processor that supports an MStream execution environment as a *Site*. The closest analogy to an MStream is a mobile active mailbox. A mailbox, like an MStream has a globally unique name. MStreams provide a *FIFO* ordering, ensuring that messages are consumed from MStream in the same order as they are sent to it. Usually mailboxes are stationary. MStreams, on the other hand, have the ability to move from Site to Site dynamically. Usually mailboxes are passive. In contrast, message arrival at an MStream can trigger the concurrent execution of message consumption event handlers (*Append Handlers*). Such handlers that are registered with the MStream process the message and can send messages to other MStreams.

An AGNI distributed system consists of one or more *Sites*. A collection of Sites participating a distributed application is called a *Session*. Each Site is assigned a *Location Identifier* that uniquely identifies it within a given Session. New Sites may be added and removed from the Session at any time. An MStream may be located on, or moved to any Site in the Session that allows it to reside there. MStreams may be opened like sockets and messages sent (*appended*) to them. Multiple event handlers may be dynamically attached, to and detached from, an MStream. Handlers are invoked on discrete changes in system state such as message delivery (append), MStream relocations, new Handler attachments new Site additions and Site failures.

Handlers can communicate with each other by *appending* messages to MStreams. These messages are delivered asynchronously to the registered Append Handlers in the same order that they were issued [3]. A message is *delivered* at an MStream when the Append Handlers of the MStream has been activated for execution as a result of the message. A message is *consumed* when all the *Append* handlers of the MStream that are activated as a result of its delivery have completed execution.

An application built on our system may be dynamically extended and re-configured in several ways while it is in execution (i.e., while there are pending un-delivered messages). First, an *Agent* can dynamically change the handlers it has registered for a given Event. Second, new Agents may be added and existing Agents (and their handlers) removed for an existing MStream. Third, new MStreams may be added and removed. Fourth, new Sites may be added and removed, and finally, MStreams may be moved dynamically from Site to Site. These changes may be restricted using resource control mechanisms that are described in greater detail in our earlier paper [13].

All changes in the configuration of an MStream, such as MStream movement, new Agent addition and deletion, and MStream destruction are deferred until the time when no Handlers of the MStream are executing. We call this the *Atomic Handler Execution Model*. Message delivery order is preserved despite dynamic reconfiguration, allowing both the sender and receiver to be in motion while asynchronous messages are pending delivery.

[3] By asynchronous delivery we mean that the sender can continue to send messages even when previously sent messages have not been consumed. Synchronous delivery of messages is supported as an option but asynchronous delivery is expected to be the common case. We do not discuss synchronous delivery in this paper

2.1 Requirements for Reliable Message Passing Between Agents

When one needs to provide reliable in-order end-to-end communications, the first issue to be addressed is ”Can we use the existing TCP ?” We argue that highly Mobile Agent systems impose unique communication requirements that are not easily addressed by conventional TCP. In particular: (1) Agents need the ability to detect and recover from failures in the end-to-end transport mechanism. If we use conventional TCP and the Site where a receiver is located fails, the sender would have no idea of what packets have been delivered. To accommodate for this, we would have to build application level protocols that address reliability beyond the existence of a single TCP connection. (2) Agents need to communicate while moving from machine to machine and TCP does not handle moving endpoints. Even in mobile TCP [1], the communication stack and the connection state remains fixed to a given machine - only the packets are re-routed as the machines move around. Mobile Agents, on the other hand are mobile applications. When the application moves, the connection has to move. (3) Agent communications need to be optimized for mobility. If we addressed mobility by creating new TCP connections to each location an Agent visited, we would suffer the connection setup cost for each move and the inability to transmit to Agents while they are in motion. In our work, we adopt many of the design features and mechanisms of TCP, but embody them in a UDP-based protocol that accommodates highly mobile end-points and extended reliability semantics and we describe our protocol in the next section.

2.2 A Sliding-window Protocol for Reliable Agent Communication

Within our AGNI framework, messages are sent to MStreams using an in-order, sender-reliable delivery scheme built on top of UDP. All messages are consumed in the order they are issued by the sender despite failures and MStream movements. When the target of a message moves, messages that have not been consumed have to be delivered to the MStream at the new Site. There are two strategies one may consider in addressing this problem (1) Forward un-consumed messages from the old Site to the new Site or (2) Re-deliver from the sender to the new Site. Forwarding messages has some negative implications for reliability and stability. If the Site from which the MStream is migrating dies before buffered messages have been forwarded to the new Site, these messages will be lost. Also, if the target MStream is moving rapidly, forwarding will result in un-consumed messages being retransmitted several times before final consumption. Hence, we followed the second strategy. In our system, the sender buffers each message until it receives notification that the handler has run and the message has been consumed, re-transmitting the message on time-out.

In our system, when an MStream moves, it takes various state information along with it. Clearly, there is an implicit movement of handler code and Agent execution state, but in addition, the MStream takes a state vector of sequence numbers. There is a slot in this vector for each live MStream that the MStream in motion has sent messages to or received messages from. Each slot contains a sent-received pair of integers indicating the next sequence number to be sent or received from a given MStream. This allows the messaging code to determine how to stamp the next outgoing message or what sequence number should be consumed next from a given sender.

Our protocol uses a sliding-window acknowledgement mechanism similar to those employed by TCP. The sender buffers un-acknowledged messages and computes a smoothed estimate of the expected round-trip time for the acknowledgment to arrive from the receiver. If the acknowledgment does not arrive in the expected time, the sender re-transmits the message. The sender keeps a transmit window of messages that have been sent but not acknowledged by the receiver and adjusts the width of this window depending upon whether an ACK was received in the expected time. The receiver keeps a reception window that it advertises in acknowledgements. The sender transmit window is the minimum of the reception window and the window size computed by the sender. As in TCP, the sender uses ACKs as a clock to strobe new packets into the network. When an MStream moves, a *Location Manager* is informed of its new location. Messages from senders that are pending delivery to the MStream that has moved use the information stored by the Location Manager to re-send messages to its new location. The slow start algorithm is adapted from the TCP and is as follows:

– Each sender has, as part of its mobile state, information pertaining to each MStream that has been opened by its Agents. This state information includes a buffer of messages that has been sent but not acknowledged, a sender congestion window for each receiver `cwnd[receiver]` and a pair of sequence numbers corresponding to the next message to send to (or receive from) a given sender (or receiver).
– When starting set `cwnd[receiver]` to 1. When the receiver has been detected as having moved by the sender, set its corresponding congestion window to 1. When a loss is detected (i.e. sender ACK does not arrive in the estimated round-trip time or a NAK arrives from the sender), halve the current value of the congestion window.

– For each new ACK from a receiver, increase the senders congestion window for the receiver (`cwnd[receiver]`) by one until the sender-established maximum or the maximum advertisement of the receiver is reached.

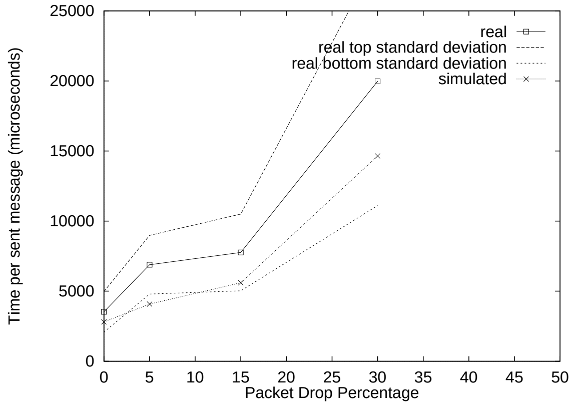
Besides integration with the Location Manager, we depart from TCP in another important regard - the receiver sends an acknowledgement for a message only after the handler has completed execution at the receiver. We adopt this strategy for two reasons (1) if the receiver moves while there are still un-consumed but buffered messages, the site from which the move originated may elect to discard these messages and the sender is responsible for retransmission and (2) if the receiver site fails before buffered messages are consumed, the sender has to detect this occurrence and re-transmit un-consumed messages.

While we have described our basic mechanism for reliability and mobility, there are several performance and scaling questions that need to be addressed at this point. First, how does the efficiency of movement notifications affect the performance of message delivery? Second, how does the receiver window get established? Third, what do we do with packets that are sent to locations where a receiver does not reside? We examine the answers to these questions in the sections that follow by using a simulation studies.

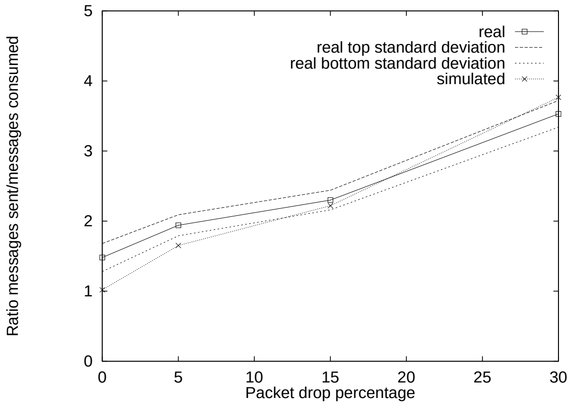
3 Simulation Approach

Estimating the detailed behavior and performance of a distributed system is difficult. There are several degrees of variability and the interaction between physical effects is often difficult to determine. In order to evaluate our algorithms under a variety of conditions, we constructed a detailed system simulation using an in-situ approach. This approach wraps a simulated environment around the actual AGNI system and application code using the CSIM [14] simulation library. We replace thread creations, semaphore locking and message sends and receives with simulated versions of these, but leave the rest of the code unmodified. We inserted simulated delays into the code at various locations and tuned these with the goal of matching real and simulated performance for various parameters of interest. One can get a good idea about which system delays are significant by looking at the gprof execution trace of the actual system. The simulation contains other parameters that have been tuned so that the simulated performance matches the actual performance of the system. We adjusted these parameters, using experimental data so that the message latency and packet drop percentage of the real and simulated systems match. We first gathered the experimental data by running the system on a test bed of machines. Next, we ran a genetic algorithm on the simulation to adjust the simulation tuning parameters so that the output of the simulation matched the actual system. We repeated this exercise over a range of scenarios to increase the confidence level in our simulation.

Figure 1 shows an example of the fit between real and simulated systems for the two quantities of interest in this paper which are : (1) the ratio of packets sent to the packets delivered which is a measure of the packets lost in transit and (2) the average time to consume a packet which is a measure of the throughput.

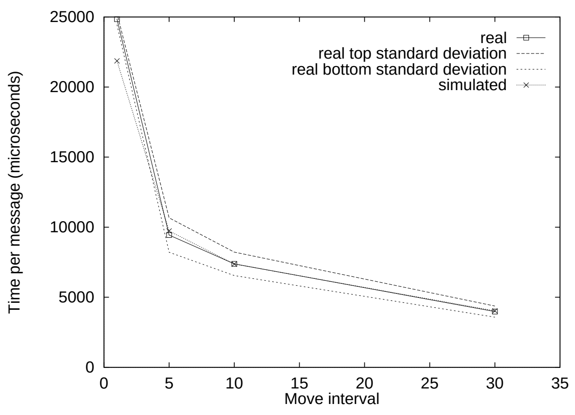


(a) Average time per message

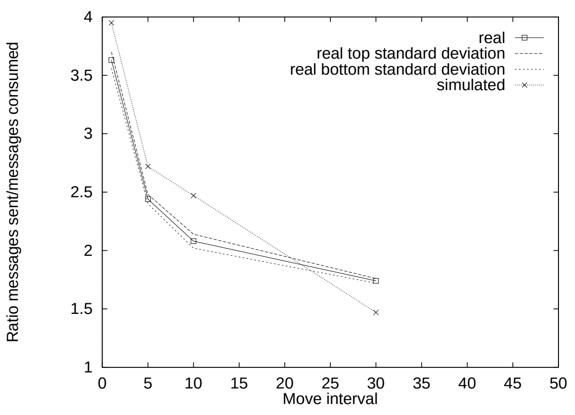


(b) Packets sent per packet consumed

Fig. 1. Simulated versus actual time per message and packet ratio. Both end-points are fixed. Packet drop is effected by random drop with uniform probability.



(a) Average time per message



(b) Packets sent per packet consumed

Fig. 2. Simulated versus actual time per message and packet ratio one end-point fixed and one end-point moving. Packet drop percentage is 0.

4 Performance Analysis of Basic Protocol Mechanisms

The reliable peer-to-peer protocol achieves its efficiency through pipelining message production with the delay caused by of the link and message consumption. It is important to characterize what happens when the system is re-configured and the pipeline is broken as a result of Agent motion. In our simulation, the system and application are characterized by : (1) **Message Production Time:** Time taken to produce a message by the sender of the message (2) **Message Consumption Time:** Time taken to consume the message by the handler at the recipient. (3) **Link Latency:** Time that the message spends "on the wire" before it reaches the recipient. (4) **Drop percentage:** To keep the model simple, we assume a uniform random drop model. Packets are dropped at random, at a fixed probability, according to a uniform distribution. (5) **Location Tracking Latency:** Delay in reporting the location of an MStream incurred by the Location Management infrastructure.

4.1 The Importance of Efficient Location Tracking

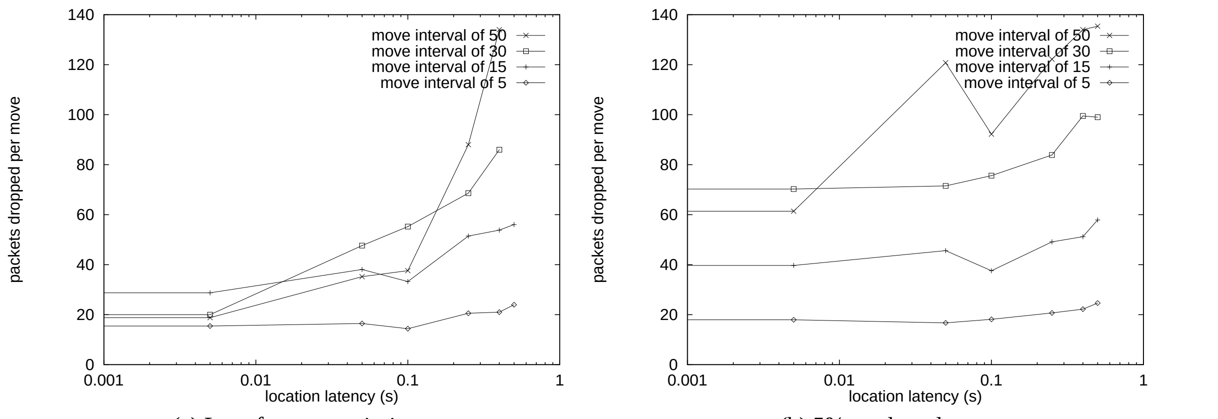
If multiple Mobile Agents are to cooperate by sending messages to each other, there has to be a means of tracking where the Agents reside so that they can find each other efficiently. The support needed for this is similar to the Domain Name Service (DNS) for IP. However, there are important considerations to be taken into account that do not present a problem for a relatively static world such as the one addressed by the DNS. A design issue that needs to be considered is propagation delay of location information. The Domain Name Service in IP is hierarchically structured which gives it great scalability. However, the hierarchical structure could imply a greater latency for answering location queries. In the context of highly mobile systems, the capabilities and the responsiveness of name-to-location mapping services are of critical importance in the design and performance of Agent communication protocols. In this section, we examine the interaction of location tracking mechanism with our protocol for reliable messaging among MStreams.

We assume that applications can be dynamically reconfigured at any time resulting in situations with both the sender and receiver moving while there are still pending, un-delivered messages. In our design, when an MStream moves, a Location Manager is informed of the new Site where the MStream will reside. This information needs to be propagated to each Agent that has opened the MStream. Three notification strategies were considered: (1) a lazy strategy where the propagation of such information is deferred and made available on a reactive basis by informing senders after their next attempt to transmit to the old location, (2) an eager strategy where the Location Manager informs all sites about the new MStream location by disseminating this information via multicast or multiple unicast messages and (3) a combined strategy which informs all senders on an eager basis and redirects misdirected messages if the need should arise.

In our protocol, the sender expects to get an acknowledgement back from the target location for a given message in 1.5 times the expected round-trip time. The estimated round-trip time is computed by keeping a running estimate of expected round trip time to each target site for each open MStream and recomputing it on each acknowledgement using a smoothing estimator similar to the one employed by TCP [6] (i.e. $RTT_i = RTT_{i-1} + \alpha * MeasuredRTT$). As in TCP, we use an α value of 0.75. The expected round-trip time includes the time for the handler to run as the acknowledgement only returns after the handler is executed at the target MStream. If the acknowledgement does not arrive in the expected time, the sender attempts a retransmission of the message. This is re-tried *k* (currently set to 3) times before the sender sends the message to the Location Manager to be forwarded to the new destination for the MStream and drops its congestion window `cwnd[receiver]` to 1. The Location Manager forwards the message to the current Site where the receiver resides and sends a notification back to the sender informing it about the new location of the receiver. Both the combined and the lazy schemes adopt this forwarding strategy. However, in the combined scheme, in addition to this forwarding service, the Location Manager informs all Sites that have an open MStream handle for the MStream in motion, immediately as soon as a movement takes place.

Location update messages can be delayed in transit, or lost. The delay in location notifications is important to consider as it affects the performance of message delivery in highly mobile systems. We have a classic case of scalability versus responsiveness and it is important to study the effect of this tradeoff to determine the cost of scalable structures. Note that we do not rely on the Sites propagating location information themselves for two reasons : (1) the Sites are assumed to be unreliable and the links are lossy so the location update message may be lost and never reach the senders (2) if multiple Sites are sending location update information, the information may be out of sync and lead to instability for rapidly moving targets.

Our first simulation scenario involves a single sender and a single receiver that is moving randomly between a set of locations. The sender sends messages to the receiver who moves after *m* messages. The receiver moves round-robin between 10 locations. Each time the receiver moves, the location manager sends a notification of the move to the senders. We examine the effect of delaying this notification on the average time to for each message and on the number of retransmissions per move. We present these results for two cases - a perfect link with no loss and a lossy link with 5% drop in the Figure 3. The interactions between the various mechanisms are a bit more complex than one might expect at the outset. For example, note that in the case of the rapidly moving endpoint (move interval of 5) the sender window never gets a chance to open up so there is no message pipeline. As the receive window never gets filled, fewer messages get dropped at the receiver. The message drop percentages are thus lower than in the other cases as no messages are dropped when the receiver moves - especially with the low location latencies.



(a) Loss free transmission (b) 5% random drop.

Fig. 3. Effect of location latency on messages dropped per move. Early messages are cached for 1 second at the receiver. The sender remains fixed. The *move interval* is the number of messages per move of the receiver.

As may be observed, with the TCP-like sender reliable protocol, location latency has a significant impact on message drop when the MStreams move frequently. When an MStream moves, the senders get notification of the new location of the MStream. If move notification to senders is delayed, the pipeline is broken for a longer period of time and hence the receiver remains idle. The sender has to transmit its full window to the new location again when the receiver has arrived. This effect is illustrated in Figure 4 which shows the effect of different propagation delays of the Location Manager on the sequence numbers of the messages sent by the sender and consumed by the receiver. The effect is as expected. A higher location latency results in the pipeline being disrupted for a longer period of time and hence greater loss and reduced performance.

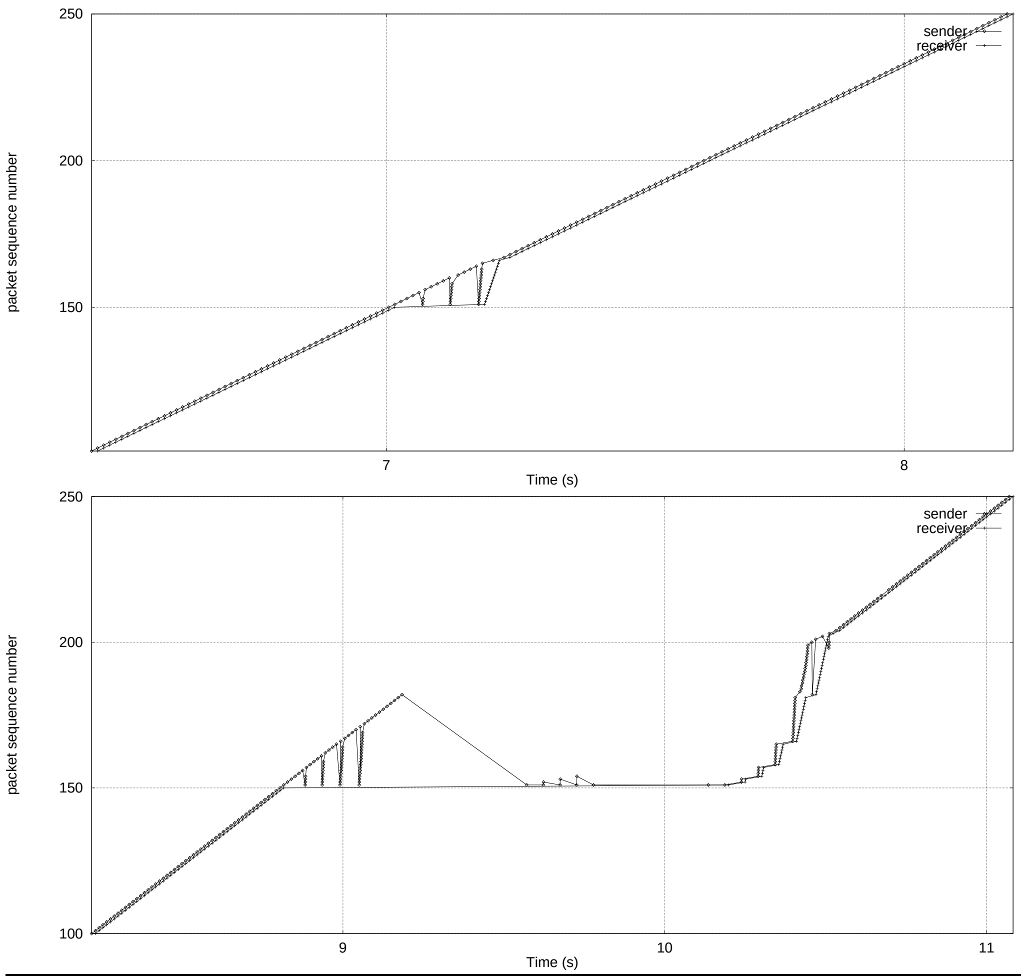
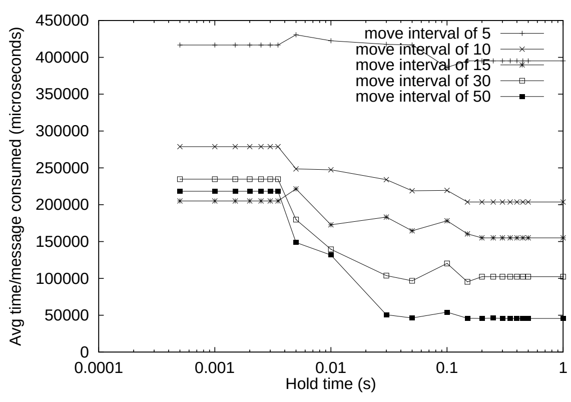


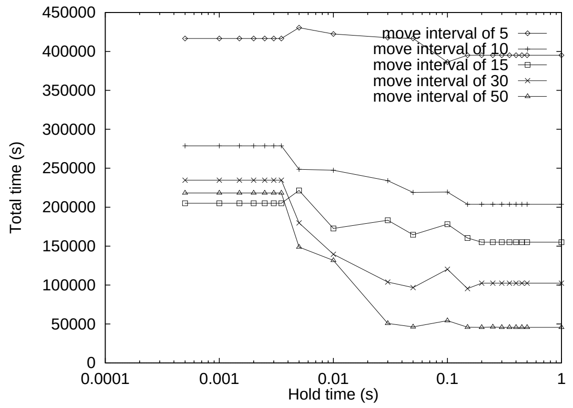
Fig. 4. The top figure shows the sequence number of sender and receiver with a location latency of 0.1 sec. and the bottom figure shows sequence numbers of sender and receiver with a location latency of 1.0 seconds. Sender is producing a message once every 0.01 seconds and the receiver handler runs for 0.001 seconds. Link latency is 0.01 seconds. The effect of increased pipeline disruption with higher location latency is evident.

4.2 Caching Mis-delivered Messages

When a move notification is received, the sender starts transmitting messages to the new location for the MStream. Note that the MStream may not have arrived at its new site by the time the sender is notified. One interesting design issue is what to do with the packets that arrive at a site prior to the arrival of the receiving MStream. The simplest strategy is to drop the packets, but in highly mobile systems, other optimizations may result in better performance. If the target Site of the move holds on to early packets for some time before discarding them, the MStream may arrive in that time and successfully consume these packets. This simple optimization has a significant effect on performance as can be seen in Figure 5. As expected, the benefit from this optimization displays a step behavior. After a threshold, holding packets for additional time does not yield greater benefit as the pipeline starvation effect caused by the move is already masked beyond this threshold.



(a) Message ratio vs. Cache hold time



(b) Average message consumption time vs. Cache hold time

Fig. 5. Opportunistic Caching - Sender is producing a message every 0.005 seconds and receiver is consuming a message every 0.001 seconds. Location latency is 0.5 seconds. The connections loss-free.

The sequence number plots of the sender and receiver shown in Figures 6 illustrate some interesting effects. In the top figure, there is no caching of packets. The pipeline shows starvation after the move. This effect can be explained by considering what happens to the send buffer right after the move. As previously described, the sender retransmits a packet 3 times to the receiver and if it does not receive an ACK in that interval it sends the packet to the location manager for forwarding and shrinks its send window down to 1. If the receiver simply discards each packet, the system goes into a state where the sender is unable to increase its send window as each successive packet reaches its maximum count and gets sent to the Location Manager for forwarding. On the other hand, if the target location holds on to the packets for a while before discarding them, this gives the receiver a chance to consume the message and send an ACK back to the sender. As can be seen from the bottom two plots in Figure 6, the pipeline is less severely disrupted by caching the packets rather than immediately discarding them after a move and hence the throughput is significantly improved.

This experiment illustrates a larger problem of stability of such a protocol. As this example shows, without damping, the system can be driven into a state of starvation. By adding buffering at the receiver, what we are doing is damping the system and this leads to quicker re-stabilization of the pipeline and a consequent improved throughput.

Our experiments indicate that a hold time of twice the expected relocation time of the receiver is an adequate hold time. In the following section, we implement a caching strategy with a hold time of 1 second which is well above this limit.

4.3 Optimizing the Receiver Window Advertisement

In TCP, the receiver advertises a window that is used to compute the sender window size. The receiver window size is the advertised buffer size for the receiver. The larger this window, the greater the possible overlap between sender, as the receiver is able to buffer these messages while the sender produces more. If the receiver moves around frequently, in our protocol, the messages in the receive window of the receiver are discarded and the receiver relies on the sender to retransmit these packets to the new site.

Figure 7 shows the effect of receiver window sizes on the packet drop and the average delivery time per packet. We kept the receiver window sizes constant for this experiment. As the plots show, it is more efficient to have a small receiver window advertisement, particularly when the receiver is moving around frequently. If the window size is set too large, the performance degrades. Our experiments indicate that a window size set equal to the expected move frequency works well and we are experimenting with a dynamic scheme where the receiver window advertisement is related to the move frequency.

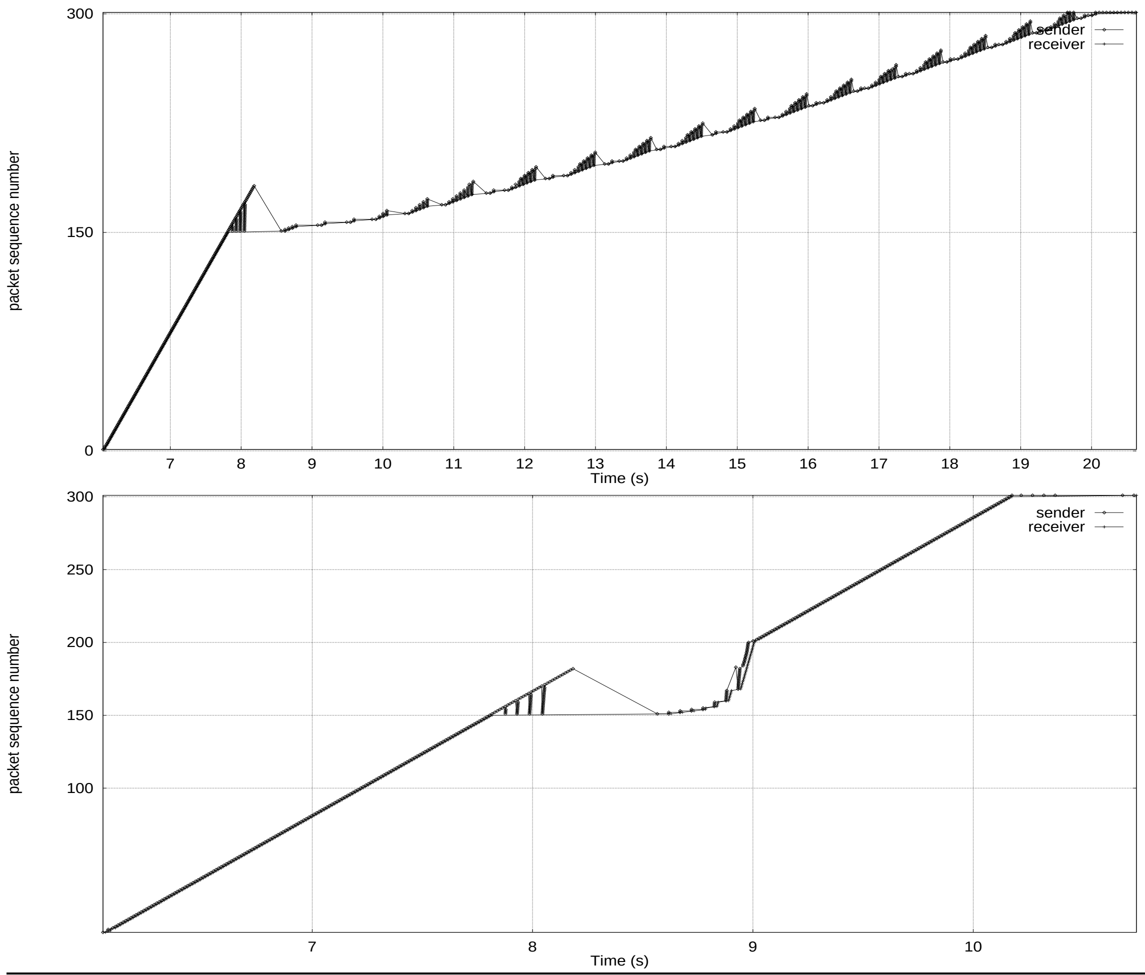
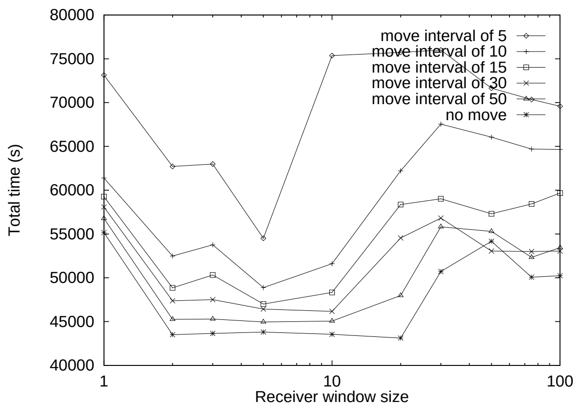
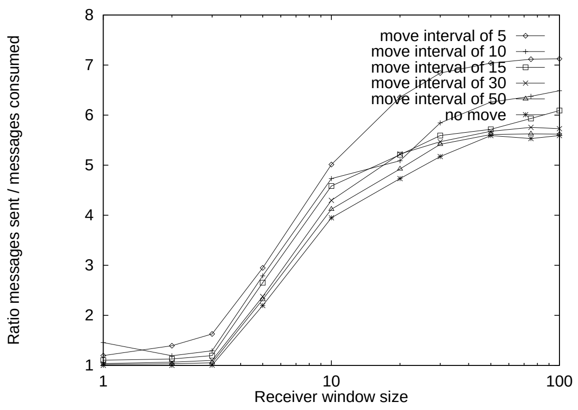


Fig. 6. The top figure shows the sequence number of sender and receiver without opportunistic caching, and the bottom figure shows the sequence number plot with 0.1 seconds cache residency. Sender is producing a message every 0.01 seconds and the receiver's append handler runs for 0.001 seconds seconds. Location latency is 0.5 seconds. Link latency is 0.01 seconds.



(a) Time per message for different receiver window sizes



(b) Message drop rate for different receiver window sizes

Fig. 7. Effect of receiver window size on throughput and message loss.

4.4 Handling Failures

Our protocol ensures that messages are delivered in order to the receiver despite failure of the site where the receiver is located. The mechanism works as follows. MStream are assigned a reliable *Failure Manager* Site. The Failure Manager has a copy of the MStream Agent code, and has knowledge of the current Sites that have opened the MStream. Failure a occurs when a Site disconnects from its Failure Manager. When a such a failure occurs each of the MStreams located at the Site that has failed are implicitly relocated to its Failure Manager Site where its *Failure Handlers* are invoked. Failures may occur and be handled at any time - including during system configuration and reconfiguration. If the Site housing an MStream should fail or disconnect while a message is being consumed or while there are messages that have been buffered and not yet delivered, re-delivery is attempted at the Failure Manager. To ensure in-order delivery in the presence of failures, the message is dequeued at the sender only after the *Append Handler*s at the receiver have completed execution and the ACK for the message has been received by the sender.

After a failure has occurred at the site where an MStream resides, a failure recovery protocol is initiated at the Failure Manager that re-synchronizes sequence numbers between all Agents that have opened the failed MStream. The sequence number vector for the failed MStream is reconstructed by querying each potential sender for its next expected sequence number. We assume that the Sites being queried may fail while this process is ongoing but that the Failure Manager site itself remains reliable.

5 Related Work

There are two other efforts in Mobile Agent communication that are closely related to our efforts. Murphy and Picco [8] present a scheme for communication in Mobile Agent systems. Our work differs in from this scheme in the following respects: (1) We have assumed that losses and failures are possible on the network. (2) We have a windowed scheme with sender-initiated retransmission for point to point messages that is fundamentally different in its operation than theirs. (3) Their scheme works for both multicast and unicast. Ours works only for unicast.

Okoshi et al. describe a mobile socket layer which is similar to our work and present details of their model in [9]. Our scheme most resembles their *Explicit Redirection* scheme. However, the focus of our work is different from theirs. They have presented a formal protocol description while we have concentrated more on performance and tuning aspects as they relate to mobility.

Mobile IP addresses the issue of global mobility [10]; that is, re-routing IP packets to nomadic machines. On this infrastructure, a reliable protocol such as mobile TCP may be built. Mobile TCP [1] addresses a different set of goals than the ones we are trying to achieve. In the case of Mobile TCP, the protocol stack remains fixed while the machine in which the protocol stack executes moves around. IP packets are redirected to the machine via mobile IP. In our case, the protocol stack itself is mobile.

In contrast with systems such as Agent Tcl [5] and Sumatra [12] we do not support strong mobility, deferring migrations to Handler boundaries. Our local event-model is very similar to commercial systems such as Agelets [7] and Voyager [4], with events being triggered by local changes in state such as message arrivals, MStream arrivals and departures. Global events are triggered by changes to the state of the distributed system. The basic mechanisms and communication abstractions that we define are sufficient to build more sophisticated mechanisms such as those supported by Mole [16, 2].

While we share several similarities with the systems mentioned above, our communication scheme differs significantly from those employed by these systems. Specifically, we do not employ forwarding pointers to redirect messages from the home location. MStreams need to rendezvous at a Site to communicate, nor do we employ RPC or rendezvous. In their examination of Mobile Agent Communication paradigms, Cabri et al. [3] argue that direct co-ordination or message passing between Mobile Agents is not advisable for the following reasons: (1) the communicating Mobile Agents need to know about each others existence (2) routing schemes may be complex and result in residual information at the visited nodes and (3) if Mobile Agents need to communicate frequently they should be co-located anyway. They suggest black-boarding style of communication for Mobile Agents. They also note that direct message passing has the advantage of being efficient and light-weight. We concur with their comments for free-roaming disconnected Agents but we have targeted our system towards distributed interacting systems rather than disconnected operations and have presented a protocol for one-way reliable message passing that we feel is appropriate and efficient for this class of applications. We agree with the observation by Cabri et al. [3], that forwarding messages when Agents are rapidly moving can result in

messages traversing the visited locations and effectively chasing the moving Agent around over several hops before being finally consumed.
Location management is an important design issue in Mobile PCS tracking. Mobile PCS location management strategies attempt to minimize the latency of locating the mobile PCS. A common scheme uses a simple two-level hierarchy with a home location that has current knowledge of the location of the user. More scalable schemes use a multi-level hierarchy [11]. As in our system, there is a trade-off between scalability and response time.
van Steen et al. [17], present a global mobile object system called *Globe* that has a scalable, hierarchical location manager. This system is suitable for slow moving objects such as those associated with people moving around from one location to another. However, Mobile Agents, can in general, move much more rapidly and as we have shown in this paper, latency of location management is an important factor communication efficiency.

6 Conclusions, Limitations and Future Work

In this paper we presented an application level, modified sliding window protocol for which functions much like TCP, but is able to handle some special requirements for Mobile Agent systems such as stack mobility, site and link failures and rapid re-configuration. We examined the effects of different location management strategies and in particular presented the effect of the latency of location resolution. Our conclusions from this study are as follows: (1) The Location Manager plays a key role in communication efficiency of a reliable protocol for Mobile Agents. For our simulations, a combined location propagation strategy worked better than a lazy one. (2) In a highly dynamic environment where Agents are frequently moving around and sending messages to each other, early messages should not be immediately discarded. We found that holding messages for several seconds for possible deferred consumption aided the protocol by reducing the number of re-transmissions. Examination of message traces revealed that this improved performance was because buffering masks the propagation of location and other information as the system is being dynamically reconfigured and allows the pipeline to continue to function while these actions are taking place. (3) Receiver window size has a significant effect on the protocol. A big receive window causes higher losses because messages in the window are discarded on move. Too small a receive window reduces throughput as no overlap is possible. A strategy that has shown promise, but that we need to examine further is to allow the receiver to set the receive window to equal the number of messages between moves. This receiver advertisement is adjusted dynamically, based on consumption rate.
It may be noted as a weakness in our approach, that that we have assumed a reliable, highly available Location Manager and Failure Manager, where some state is centralized. Both the Location Manager and the Failure Manager may be replicated for reliability and the messaging protocol remains unchanged; however, there would be additional protocol complexity to deal with failures of these Sites themselves when failure handling and motion is taking place. The strategy of sending an acknowledgement only after the Append Handler at the receiver has completed execution has a possible shortcoming; i.e. if the time for for which the append handler runs shows a great degree of variability, the retransmit timer of the sender will not converge. We are assuming that the Append Handler that runs on message consumption runs for a finite interval of time and does not show extreme variability, which are reasonable assumptions for the of the types of reactive distributed test scripting environments for which we are targeting our system, but may not hold for other systems.
We have also implemented a simulation of a simplified version of Scalable Reliable Multicast for Mobile Agents. We defer presenting the details of this protocol to a future work. We hope that our implementation and simulation environment will be useful for others designing algorithms and applications for Mobile Agent systems and we make it available for public download from *http://www.antd.nist.gov/itg/agni* or via email.

7 Acknowledgments

We acknowledge and appreciate the contributions of Laurent Andrey of LORIA (Fr.) to the design and implementation of AGNI. We are grateful to the anonymous reviewers and to the NIST internal reviewers for comments and suggestions contributing to the improvement of this paper.

References

1. Ajay Bakre and B. R. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. Technical Report DCS-TR-314, Rutgers University, October 1994.

2. Joachim Baumann, Fritz Hohl, Nikolaos Radouniklis, Kurt Rothermel, and Markus Strasser. Communication Concepts for Mobile Agent Systems. In *First International Workshop on Mobile Agents*, Berlin, Germany, April 1997.
3. G. Cabri, L. Leornardi, and F. Zambonelli. Coordination in Mobile Agent Systems. Technical Report DSI-97-24, Universita' di Modena, October 1997.
4. Object Space Corp. Voyager White Paper. http://www.objectspace.com/voyager.
5. Robert S. Gray. Agent TCL: A Flexible and Secure Mobile-agent System. In *Proceedings of the Fourth Annual Tcl/Tk Workshop - Monterey CA*, July 1996. http://www.cs.dartmouth.edu/ agent/papers.html.
6. Van Jacobson. Congestion Avoidance and Control. In *Proceedings ACM SIGCOMM*, pages 157–173, August 1988.
7. Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998. ISBN 0-201-32582-9.
8. Amy Murphy and Gian Pietro Picco. Reliable Communication for Highly Mobile Agents. In *Agent Systems and Architectures/Mobile Agents (ASA/MA) '99*, pages 141–150, October 1999.
9. Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hideyuki Tokuda. MobileSocket: Session Layer Continuous Operation Support for Java Applications. *Transactions of Information Processing Society of Japan*, 1(1), 1999.
10. Charles Perkins. RFC 2002 IP Mobility Support. See IETF RFC Repository http://www.ietf.org.
11. E. Pitoura and G. Samara. Locating Objects in Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering*, to appear(to appear), 2000. http://zeus.cs.uoi.gr/ pitoura/.
12. M. Ranganathan, Anurag Acharya, Shamik Sharma, and Joel Saltz. Network-aware Mobile Programs. In *USENIX Winter Technical Conference*, jan 1997.
13. M. Ranganathan, V. Schaal, V. Galtier, and D. Montgomery. Mobile Streams: A Middleware for Reconfigurable Distributed Scripting. In *Agent Systems And Architectures/Mobile Agents '99* , October 1999.
14. Mesquite Software. Csim-18 simulation library. http://www.mesquite.com.
15. W. Richard Stevens. *TCPIP Illustrated, Vol 1: The Protocols*. Addison-Welsley, Reading, MA, 1994.
16. Markus Strasser, Joachim Baumann, and Fritz Hohl. Mole - A Java-based Mobile Agent System. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 28–35, Linz, Austria, July 1996. http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1996-strasser-01.ps.gz.
17. M. van Steen, P. Homburg, and A.S. Tanenbaum. The Architectural Design of Globe: A Wide-area Distributed System. Technical Report IR-422, Vrije University, March 1997.